Problem statement:

1. Extract operand name
2. Rename operand
3. Hash operand in operation and binary operation to find redundancy

Function support:

1. Function1: extract address/intermediate value/result operand from all the instructions
   With llvm getOperand() function, we can only extract %a or %a.addr in an instruction, with intermediate value or register name, we have to use raw_string_ostream to extract the name of the operand, here we use a function called getOperandname() to extract all the information in all the instructions. This function will be called in store/binary function. In the load function, we only have to use inst.getOperand(0)->getName() to get the load address name, and we can assign the register sequentially. Function renameaddress() is used for simplicity, which is not necessary.

```
string getOperandname(Value* val) {
  size_t regPos;
    string Operandname;
    raw_string_ostream raw_string(Operandname);
    raw_string << *val;
    string size_string = "i32";
    if((regPos = Operandname.find_first_of("%")) != string::npos){
      Operandname = Operandname.substr(regPos,2);
    }
    else if ((regPos = Operandname.find(size_string))!= string::npos) {
      Operandname = Operandname.substr(regPos+4,Operandname.size());
    }
    // errs()<<Operandname<<"\n";
    return Operandname;
}
```

```
    if(inst.getOpcode() == Instruction::Load){
        // errs() << "This is Load"<<"\n";
        // errs() << "Load: " << inst.getOperand(0)->getName()<<"\n";
    string loadFromOp = renameaddress(inst.getOperand(0)->getName());
        //  errs() << "----------Load--------------"<<"\n";

    // errs()<<loadFromOp<<"\n";
    string reg_name = "%"+std::to_string(reg_count);
    // errs()<<reg_name<<"\n";
    reg_count++;
    op_map[reg_name] = op_map[loadFromOp];
    errs()<<inst<<"\t"<<op_map[reg_name]<<" = "<<op_map[reg_name]<<"\n";
    // errs()<<"register rename\t"<< reg_name<< "\t" <<op_map[reg_name]<<"\n";
    }
```

2. Function2: hash function for all the instructions
   a. In Store instruction, we hash the source and destination of the instruction with different key share the same value:

```
if(inst.getOpcode() == Instruction::Store){
    // errs() << "This is Store"<<"\n";
    //  errs() << "----------Store---------------"<<"\n";

    string storeSrc = inst.getOperand(0)->getName();
    //if constant, has no name
    if (storeSrc == "") {

        storeSrc= getOperandname(inst.getOperand(0));
    }
    else{
      // errs()<<"not blanck for storeSrc"<<"\n";
    }

    //storeDest is always a variable (never a constant)
    string storeDest = renameaddress(inst.getOperand(1)->getName());
    // errs()<<storeSrc<<"\n";
    // errs()<<storeDest<<"\n";

    if(storeSrc==storeDest){
        op_map[storeSrc] = op_name;
        op_name++;
    }
    else{
      if (op_map.find(storeSrc) == op_map.end())
        {
          op_map[storeSrc] = op_name;
          op_map[storeDest] = op_name;

          op_name++;

        }
      else
        {
          op_map[storeDest] = op_map[storeSrc];
        }
    }
    errs()<<inst<<"\t"<<op_map[storeDest]<<" = "<<op_map[storeSrc]<<"\n";
      // errs() << storeSrc << "\t"<<op_map[storeSrc]<<"\n";
      // errs() << storeDest << "\t"<<op_map[storeDest]<<"\n";
```

b. In Load function, we has the source operand and destination register with different key share the same value

```
if(inst.getOpcode() == Instruction::Load){
    // errs() << "This is Load"<<"\n";
    // errs() << "Load: " << inst.getOperand(0)->getName()<<"\n";
string loadFromOp = renameaddress(inst.getOperand(0)->getName());
    //  errs() << "----------Load---------------"<<"\n";

// errs()<<loadFromOp<<"\n";
string reg_name = "%"+std::to_string(reg_count);
// errs()<<reg_name<<"\n";
reg_count++;
op_map[reg_name] = op_map[loadFromOp];
errs()<<inst<<"\t"<<op_map[reg_name]<<" = "<<op_map[reg_name]<<"\n";
// errs()<<"register rename\t"<< reg_name<< "\t" <<op_map[reg_name]<<"\n";
}
```

c. In Binary operation, we first load the value for both operand from the op_map hash table, then we use the hash value to make an new binary operation, with this new binary operation, we can check it in the operation hash table to find whether it exists before to find the redundancy in the whole program.

3. Result showing:
   By running ./run.sh we can get the result like this:

```
ValueNumbering: test3
  store i32 %a, i32* %a.addr, align 4   1 = 1
  store i32 %c, i32* %c.addr, align 4   2 = 2
  store i32 %d, i32* %d.addr, align 4   3 = 3
  store i32 %x, i32* %x.addr, align 4   4 = 4
  store i32 %y, i32* %y.addr, align 4   5 = 5
  store i32 %z, i32* %z.addr, align 4   6 = 6
  %0 = load i32, i32* %x.addr, align 4  4 = 4
  %1 = load i32, i32* %y.addr, align 4  5 = 5
  %add = add nsw i32 %0, %1     7 = 4 add 5
  store i32 %add, i32* %a.addr, align 4 7 = 7
  %2 = load i32, i32* %y.addr, align 4  5 = 5
  store i32 %2, i32* %z.addr, align 4   5 = 5
  store i32 17, i32* %d.addr, align 4   8 = 8
  %3 = load i32, i32* %x.addr, align 4  4 = 4
  %4 = load i32, i32* %z.addr, align 4  5 = 5
  %add1 = add nsw i32 %3, %4    7 = 4 add 5     (redundant)
  store i32 %add1, i32* %c.addr, align 4        7 = 7
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

ValueNumbering: test4
  store i32 %a, i32* %a.addr, align 4   1 = 1
  store i32 %b, i32* %b.addr, align 4   2 = 2
  store i32 %c, i32* %c.addr, align 4   3 = 3
  store i32 %d, i32* %d.addr, align 4   4 = 4
  %0 = load i32, i32* %b.addr, align 4  2 = 2
  %1 = load i32, i32* %c.addr, align 4  3 = 3
  %add = add nsw i32 %0, %1     5 = 2 add 3
  store i32 %add, i32* %a.addr, align 4 5 = 5
  %2 = load i32, i32* %a.addr, align 4  5 = 5
  %3 = load i32, i32* %d.addr, align 4  4 = 4
  %sub = sub nsw i32 %2, %3     6 = 5 sub 4
  store i32 %sub, i32* %b.addr, align 4 6 = 6
  %4 = load i32, i32* %b.addr, align 4  6 = 6
  %5 = load i32, i32* %c.addr, align 4  3 = 3
  %add1 = add nsw i32 %4, %5    7 = 6 add 3
  store i32 %add1, i32* %c.addr, align 4        7 = 7
  %6 = load i32, i32* %a.addr, align 4  5 = 5
  %7 = load i32, i32* %d.addr, align 4  4 = 4
  %sub2 = sub nsw i32 %6, %7    6 = 5 sub 4     (redundant)
  store i32 %sub2, i32* %d.addr, align 4        6 = 6
```

```
ValueNumbering: test
  store i32 %a, i32* %a.addr, align 4   1 = 1
  store i32 %b, i32* %b.addr, align 4   2 = 2
  %0 = load i32, i32* %a.addr, align 4  1 = 1
  %1 = load i32, i32* %b.addr, align 4  2 = 2
  %add = add nsw i32 %0, %1     3 = 1 add 2
  store i32 %add, i32* %c, align 4      3 = 3
  %2 = load i32, i32* %b.addr, align 4  2 = 2
  %3 = load i32, i32* %c, align 4       3 = 3
  %add1 = add nsw i32 %2, %3    4 = 2 add 3
  store i32 %add1, i32* %d, align 4     4 = 4
  %4 = load i32, i32* %a.addr, align 4  1 = 1
  %5 = load i32, i32* %b.addr, align 4  2 = 2
  %add2 = add nsw i32 %4, %5    3 = 1 add 2     (redundant)
  store i32 %add2, i32* %e, align 4     3 = 3
  %6 = load i32, i32* %b.addr, align 4  2 = 2
  %7 = load i32, i32* %e, align 4       3 = 3
  %add3 = add nsw i32 %6, %7    4 = 2 add 3     (redundant)
  store i32 %add3, i32* %f, align 4     4 = 4
  %8 = load i32, i32* %f, align 4       4 = 4
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.
```

```
ValueNumbering: test1
  store i32 %a, i32* %a.addr, align 4   1 = 1
  store i32 %b, i32* %b.addr, align 4   2 = 2
  store i32 %c, i32* %c.addr, align 4   3 = 3
  store i32 %d, i32* %d.addr, align 4   4 = 4
  store i32 %e, i32* %e.addr, align 4   5 = 5
  store i32 %f, i32* %f.addr, align 4   6 = 6
  store i32 %g, i32* %g.addr, align 4   7 = 7
  %0 = load i32, i32* %a.addr, align 4  1 = 1
  %1 = load i32, i32* %b.addr, align 4  2 = 2
  %add = add nsw i32 %0, %1     8 = 1 add 2
  store i32 %add, i32* %c.addr, align 4 8 = 8
  %2 = load i32, i32* %c.addr, align 4  8 = 8
  %add1 = add nsw i32 %2, 5     10 = 8 add 9
  store i32 %add1, i32* %d.addr, align 4        10 = 10
  %3 = load i32, i32* %a.addr, align 4  1 = 1
  %4 = load i32, i32* %b.addr, align 4  2 = 2
  %add2 = add nsw i32 %3, %4    8 = 1 add 2     (redundant)
  store i32 %add2, i32* %e.addr, align 4        8 = 8
  %5 = load i32, i32* %e.addr, align 4  8 = 8
  %add3 = add nsw i32 %5, 5     10 = 8 add 9    (redundant)
  store i32 %add3, i32* %f.addr, align 4        10 = 10
  %6 = load i32, i32* %d.addr, align 4  10 = 10
  %7 = load i32, i32* %f.addr, align 4  10 = 10
  %add4 = add nsw i32 %6, %7    11 = 10 add 10
  store i32 %add4, i32* %g.addr, align 4        11 = 11
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

ValueNumbering: test2
  store i32 %a, i32* %a.addr, align 4   1 = 1
  store i32 %b, i32* %b.addr, align 4   2 = 2
  store i32 %c, i32* %c.addr, align 4   3 = 3
  store i32 %x, i32* %x.addr, align 4   4 = 4
  store i32 %y, i32* %y.addr, align 4   5 = 5
  %0 = load i32, i32* %x.addr, align 4  4 = 4
  %1 = load i32, i32* %y.addr, align 4  5 = 5
  %add = add nsw i32 %0, %1     6 = 4 add 5
  store i32 %add, i32* %b.addr, align 4 6 = 6
  %2 = load i32, i32* %x.addr, align 4  4 = 4
  %3 = load i32, i32* %y.addr, align 4  5 = 5
  %add1 = add nsw i32 %2, %3    6 = 4 add 5     (redundant)
  store i32 %add1, i32* %a.addr, align 4        6 = 6
  store i32 17, i32* %a.addr, align 4   7 = 7
  %4 = load i32, i32* %x.addr, align 4  4 = 4
  %5 = load i32, i32* %y.addr, align 4  5 = 5
  %add2 = add nsw i32 %4, %5    6 = 4 add 5     (redundant)
```

```
ValueNumbering: test5
  store i32 %a, i32* %a.addr, align 4   1 = 1
  store i32 %b, i32* %b.addr, align 4   2 = 2
  store i32 %c, i32* %c.addr, align 4   3 = 3
  store i32 %d, i32* %d.addr, align 4   4 = 4
  store i32 %e, i32* %e.addr, align 4   5 = 5
  %0 = load i32, i32* %b.addr, align 4  2 = 2
  %1 = load i32, i32* %c.addr, align 4  3 = 3
  %mul = mul nsw i32 %0, %1     6 = 2 mul 3
  store i32 %mul, i32* %a.addr, align 4 6 = 6
  %2 = load i32, i32* %b.addr, align 4  2 = 2
  store i32 %2, i32* %d.addr, align 4   2 = 2
  %3 = load i32, i32* %a.addr, align 4  6 = 6
  %4 = load i32, i32* %b.addr, align 4  2 = 2
  %add = add nsw i32 %3, %4     7 = 6 add 2
  store i32 %add, i32* %c.addr, align 4 7 = 7
  %5 = load i32, i32* %d.addr, align 4  2 = 2
  %6 = load i32, i32* %c.addr, align 4  7 = 7
  %mul1 = mul nsw i32 %5, %6    8 = 2 mul 7
  store i32 %mul1, i32* %e.addr, align 4        8 = 8
  %7 = load i32, i32* %c.addr, align 4  7 = 7
  %add2 = add nsw i32 %7, 5     10 = 7 add 9
  store i32 %add2, i32* %d.addr, align 4        10 = 10
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

ValueNumbering: test6
  store i32 %c, i32* %c.addr, align 4   1 = 1
  store i32 %d, i32* %d.addr, align 4   2 = 2
  store i32 %e, i32* %e.addr, align 4   3 = 3
  store i32 %i, i32* %i.addr, align 4   4 = 4
  store i32 %j, i32* %j.addr, align 4   5 = 5
  store i32 10, i32* %a, align 4        6 = 6
  store i32 40, i32* %b, align 4        7 = 7
  %0 = load i32, i32* %i.addr, align 4  4 = 4
  %1 = load i32, i32* %j.addr, align 4  5 = 5
  %mul = mul nsw i32 %0, %1     8 = 4 mul 5
  store i32 %mul, i32* %t1, align 4     8 = 8
  %2 = load i32, i32* %t1, align 4      8 = 8
  %add = add nsw i32 %2, 40     9 = 8 add 7
  store i32 %add, i32* %c.addr, align 4 9 = 9
  store i32 150, i32* %t2, align 4      10 = 10
  %3 = load i32, i32* %c.addr, align 4  9 = 9
  %mul1 = mul nsw i32 150, %3   11 = 10 mul 9
  store i32 %mul1, i32* %d.addr, align 4        11 = 11
  %4 = load i32, i32* %i.addr, align 4  4 = 4
  store i32 %4, i32* %e.addr, align 4   4 = 4
  %5 = load i32, i32* %i.addr, align 4  4 = 4
  %6 = load i32, i32* %j.addr, align 4  5 = 5
  %mul2 = mul nsw i32 %5, %6    8 = 4 mul 5     (redundant)
  store i32 %mul2, i32* %t3, align 4    8 = 8
  %7 = load i32, i32* %i.addr, align 4  4 = 4
  %mul3 = mul nsw i32 %7, 10    12 = 4 mul 6
  store i32 %mul3, i32* %t4, align 4    12 = 12
  %8 = load i32, i32* %t1, align 4      8 = 8
  %9 = load i32, i32* %t4, align 4      12 = 12
  %add4 = add nsw i32 %8, %9    13 = 8 add 12
```

```
if (inst.isBinaryOp())
{
    // errs() << "Op Code:" << inst.getOpcodeName()<<"\n";
    //  errs() <<"--------"<<inst.getOpcodeName()<<"--------"<<"\n";
      // size_t count = 1;
      // vector <string> operator_vec;
    bool repeat = false;

    operation = inst.getOpcodeName();
    // auto item=operator_vec.begin();
    // std::vector<string>::iterator item;
    //  errs() <<"--------"<<operation<<"--------"<<"\n";

    if ( std::find(operator_vec.begin(), operator_vec.end(), operation) != operator_vec.end() ){
      result_op = operation+to_string(count);
      count++;

    }
      // do_this();
    else{
      operator_vec.push_back(operation);
      result_op = operation;
      // errs()<<operator_vec[0]<<"\n";
      // errs() << "First time "<<operation<<"\n";
    }
    //  errs() <<"result op \t"<< result_op<<"\n";
```

```
      binary_op = to_string(op_map[op0])+operation+to_string(op_map[op1]);
      // errs()<<binary_op<<"\n";
        if (value_map.find(binary_op) == value_map.end())
          {
            value_map[binary_op] = result_op;
            // op_map[storeDest] = op_name;
            op_map[result_op] = op_name;
            op_name++;

            // value_name++;

          }
        else
          {
            // value_map[binary_op] = op_map[storeSrc];
            op_map[result_op] = op_map[value_map[binary_op]];
            repeat = true;
            // errs()<<"redundant"<<"\n";

          }
        // errs()<<result_op<<"\t"<<op_map[result_op]<<"\n";
        errs()<<inst<< "\t"<<op_map[result_op]<<" = "<< to_string(op_map[op0])+ " " + operation+ " " +to_string(op_map[op1]);
        if(repeat){
          errs()<<"\t"<<"(redundant)"<<"\n";
        }
        else{
          errs()<<"\n";
        }
```